A Specification of TCP/IP using Mixed Intuitionistic Linear Logic (Extended Abstract)

David Gray*, Geoff Hamilton*, James Power†, David Sinclair*
*School of Computer Applications
Dublin City University
Glasnevin, Dublin 9, Ireland
†Department of Computer Science
National University of Ireland, Maynooth
Maynooth, Co. Kildare, Ireland

15 March 2001

1 Introduction

This paper presents an outline specification of the IP and TCP communication protocols in mixed intuitionistic linear logic and describes how this logic can be used to prove some properties of both protocols. We have previously presented a specification of IP in [8] using commutative linear logic; in this paper we extend this specification considerably to include TCP, which, in turn, necessitates the use of non-commutative operators.

Linear logic is particularly suited to the description of state-based systems, and communication protocols in particular, since it keeps track of the resources used in each deduction step. Mixed intuitionistic linear logic is variant of linear logic that contains both commutative and non-commutative operators, and as such is useful where the order of the consumption of resources must be specified.

In the following sections we briefly describe IP and TCP and linear logic. We then present an outline of our specification of the user interfaces for IP and TCP, demonstrating the role of the linear operators in the axioms. Finally, we outline the remainder of our work concerning the description of the TCP protocol itself and the verification process undertaken.

1.1 TCP/IP

The Transmission Control Protocol (TCP)[7] and the Internet Protocol (IP)[6] are two essential elements of the communications stack, at the heart of many network-based applications. Both of these protocols are typical of state-based distributed systems. IP is responsible for transmitting data from one internet node to another, but does not guarantee the delivery of data to the destination node. TCP is a protocol that sits on top of IP and it has the responsibility of establishing an end-to-end error free connection between peer TCP entities.

IP provides no mechanisms to provide end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host communications protocols. There are no acknowledgements either end-to-end or hop-by-hop, and the error control only covers the IP packet header and not the data itself. In IP there is no flow control or retransmission. IP packets can be lost, duplicated and delivered in any order.

It is the function of TCP to establish an error-free end-to-end connection between peer TCP entities. Since IP provides no guarantees in relation to data delivery, TCP provides all the necessary mechanisms, such as flow control, acknowledgements, and retransmission, to ensure that the data is delivered in sequence and without duplication or error. TCP accomplishes this by segmenting the data and associating unique sequence numbers with each segment.

1.2 Mixed Intuitionistic Linear Logic

Linear logic [3] belongs to the family of sub-structural logics, which modify or eliminate the usual structural rules of Contraction, Exchange and Weakening. With linear logic the Contraction rule, which allows hypothesis to be duplicated, and the Weakening rule, which allows hypothesis to be discarded, are removed. The effect of this is to make the logic "resource conscious", since each step in a deduction can be regarded in terms of its consumption or production of logical statements.

Accepted for the 2nd Joint Workshop on Formal Specification of Computer-Based Systems
Washington DC, USA, 20 April, 2001

Further, non-commutative linear logic [1] removes an another structural rule, Exchange, which allows hypotheses to be reordered. Since this can be too restrictive in general, mixed intuitionistic linear logic (MILL) [2] combines both commutative and non-commutative logics in the one system. The removal of the structural rules means that the ordinary logical operators for conjunction, disjunction and implication are replaced with linear and non-commutative versions. These include:

- **Multiplicative conjunction** is written as $A \otimes B$. When used both hypothesis are consumed in any order and are no longer available.
- **Non-commutative multiplicative conjunction** is written as $A \odot B$ and represents the consumption of B after A.
- **Additive conjunction** is written $A \& B$. When used it represents a deterministic choice of the hypothesis to be consumed.
- **Additive disjunction** is written $A \oplus B$. When used it represents an external (non-deterministic) choice as to the hypothesis to be consumed.
- **Linear implication** is written $A \multimap B$, representing a process that consumes A and produces B.
- **Direct implication** is written $A \bullet\!\!- B$. If you have two hypotheses A and $A \bullet\!\!- B$ then you can derive B provided the hypotheses A are consumed in order.

The basis of our specification consists of a series of axioms, presented using the linear operators, which specify the valid transitions that can take place in the system. For IP we use ordinary commutative linear logic, since IP datagrams may be reordered in transmission. However, since order of receipt is important for TCP, we make use of a combination of commutative and non-commutative operators in its specification.

2 Outline of the Specification

In this section we briefly outline the main axioms that describe both the IP and TCP user interfaces. It should be noted that the full specification also involves a description of the TCP protocol which links these interfaces, but these axioms has been elided in this abstract.

2.1 The Internet Protocol User Interface

There are two main operations available to the user of the IP layer:

- $Send(x, y, ttl, m)$
  Sends a message m from node x to node y with a "time to live" value of ttl.
- $Rcv(x, y, ttl, m)$
  Receives a message m from node x to node y with a "time to live" value of ttl.

We use three axioms to define the operation of the IP layer, describing the sending of datagrams, their possible loss or duplication during transmission, and their receipt.

First, sending a message adds a single datagram to the system.

$$\begin{array}{l} \forall x.\forall y.\forall ttl.\forall m. \\ Send(x, y, ttl, m) \quad \multimap \quad Datagram(x, y, lower(ttl), m) \end{array} \tag{1}$$

Here $lower$ is a function that reduces its operand to some non-zero integer in the range $(0, ttl]$.

Second, when in transmission a datagram can be duplicated or lost.

$$\begin{array}{l} \forall x.\forall y.\forall ttl.\forall m. \\ Datagram(x, y, ttl, m) \quad \multimap \quad (Datagram(x, y, lower(ttl), m) \otimes Datagram(x, y, lower(ttl), m)) \\ \qquad\qquad\qquad\qquad\quad \oplus \quad \mathbf{1} \end{array} \tag{2}$$

1 is the unit for multiplicative conjunction, and is commonly used to represent the consumption of resources with no corresponding product.

Finally, if a datagram addressed to node y exists and node y is listening for it, the node y will receive the message m or some corrupted version m' of that message.

$$\begin{array}{l} \forall x.\forall y.\forall ttl.\forall m. \\ Datagram(x,y,ttl,m) \otimes Listen(y) \quad \multimap \quad Listen(y) \quad \otimes \\ \qquad\qquad\qquad\qquad\qquad\qquad (Rcv(x,y,lower(ttl),m) \oplus Rcv(x,y,lower(ttl),m')) \end{array} \tag{3}$$

Of course many issues relevant to IP have been omitted here (most notably routing and fragmentation), but this specification provides a sufficient base for the verification of the relevant properties of TCP.

2.2 TCP User Operations

We will define the user interface to the TCP layer as consisting of the following operations:

- $New(s,d)$
  Create a socket for use in making a connection between a source internet address s and a destination address d.
- $Accept(s,d)$
  Accept an attempt from some internet address s to make a connection to the internet address d.
- $Write(s,d,m)$
  Write the data m on the connection from internet address s to internet address d.
- $Read(s,d,l)$
  Attempt to read l octets from a connection from internet address s to internet address d. If there is less than l octets in the connection, *Read* will read all the octets in the connection.
- $Close(s,d)$
  Terminate the connection between internet addresses s and d.

In order to describe the TCP user interface we define a predicate to describe the current status of a connection.

- $Stream(s,d,m_w,m_r,b)$
  This represents one side of the connection from address s to address d. The data written by s is split into m_w, the data not yet read by d, and m_r, the data that *has* been read by d. The variable b is a boolean flag which is false once an end of stream (EOS) character is written to the stream.

A full TCP session will thus consists of a pair of these streams, one each for s and d.

A connection is set-up between internet addresses s and d if both of the hosts of internet addresses s and d actively specify the connection or if one host actively specifies the connection and the other host passively accepts connections from a specified internet address.

$$\begin{array}{l} \forall s.\forall d. \\ \left(\begin{array}{l} (New(s,d) \otimes New(d,s)) \oplus \\ (Accept(s,d) \otimes New(d,s)) \end{array}\right) \quad \multimap \quad Stream(s,d,\mathbf{nil},\mathbf{nil},\mathbf{true}) \otimes Stream(d,s,\mathbf{nil},\mathbf{nil},\mathbf{true}) \end{array} \tag{4}$$

Axiom 5 specifies the effect of writing some data at s, which is then appended to the data previously written (we use "::" to represent list concatenation).

$$\begin{array}{l} \forall s.\forall d.\forall m.\forall m_1.\forall m_2 \\ Stream(s,d,m_1,m_2,\mathbf{true})) \odot Write(s,d,m) \quad \multimap \quad Stream(s,d,m_1{::}m,m_2,\mathbf{true}) \end{array} \tag{5}$$

Axioms 6 and 7 specify the operation of reading some data at d, which transfers exactly l octets from m_1 to m_2.

$$\begin{array}{l}\forall s.\forall d.\forall m.\forall m_1.\forall m_2.\forall b.\forall l = \text{length}(m). \\ Stream(s,d,m{::}m_1,m_2,b) \odot Read(s,d,l) \quad \multimap\!\bullet \quad Stream(s,d,m_1,m_2{::}m,b)\end{array} \tag{6}$$

$$\begin{array}{l}\forall s.\forall d.\forall m_1.\forall m_2.\forall b.\forall l > \text{length}(m_1). \\ Stream(s,d,m_1,m_2,b) \odot Read(s,d,l) \quad \multimap\!\bullet \quad Stream(s,d,\mathbf{nil},m_2{::}m_1,b)\end{array} \tag{7}$$

Finally, axiom 8 specifies the normal closing sequence, where it is only necessary to change the flag on the relevant stream from *true* to $false$. Note that the premises in equations 5, 6 and 7 will ensure that this stream can still be read from, but not written to.

$$\begin{array}{l}\forall s.\forall d.\forall m_1.\forall m_2 \\ Close(s,d) \otimes Stream(s,d,m1,m2,\mathbf{true}) \quad \multimap \quad Stream(s,d,m1,m2,\mathbf{false})\end{array} \tag{8}$$

We define a *valid* TCP connection as one consisting of two streams, where the data read at one address is exactly the data written by the other. Specifying this inductively within the system is straightforward: a valid session is any sequence of linear propositions (including, of course, the commands defined above) that gives rise to a consistent, completed pair of streams.

3 Conclusions and Further Work

The above specification only presents an outline of the work. In particular, we have elaborated this work in three directions, as follows.

- **Specifying the TCP Data Transfer Protocol**

  The TCP user interface of the previous section represented each direction of the data transfer using a single predicate. At the protocol level we represent this state using two predicates, one each for the sender and receiver. The task of the TCP protocol specification is to ensure that these predicates, while maintained separately, are kept in a consistent state by each of the protocol axioms. Keeping track of the sequence numbers of the data sent and received will be central to this task.

  The two predicates representing a stream of data from s to d are:

  – $SendState(s,d,s\_una,s\_nxt,wbuf,rtq,r\_wnd)$
  This is the state at s of data transfer to d. Here, s_una is the oldest unacknowledged sequence number, s_nxt is the next sequence number to be sent, and r_wnd is the current size of the receiver window. The write buffer $wbuf$ holds data written by s but not yet sent, while the retransmission queue, rtq, holds data that has been sent, but not yet acknowledged.

  – $RecvState(d,s,r\_nxt,r\_wnd,rbuf)$
  This is the state at d on a stream receiving data from s. The sequence number r_nxt is the start sequence number of the next data to be received. Once received, the data will be stored in $rbuf$, where the size of the receive window, r_wnd, indicates the available space left in this buffer.

  In addition we make use of two predicates $HaveWritten$ and $HaveRead$ to record all the data written by the sender and read by the user respectively. These are necessary in order to state the basic consistency property of the TCP protocol - that all the data written by the sender is eventually read, in the same order, by the receiver.

  In order to fully specify this protocol a further 11 axioms are required which will be presented in the full version of this paper.

- **Verification of the Specification**

  The specification was verified for correctness by formulating an invariant that was maintained by each of these 11 axioms. In the final state where all data has been sent, preservation of the invariant would imply ordered transmission and receipt of all the data. The approach taken here was to express each of the data structures involved in the sender's and receiver's state as a subsequence of the total data being transmitted, indexed by the sequence numbers.

  Each piece of data written (as captured by the $HaveWritten$ predicate) should reside either in the write buffer, the retransmission queue, the read buffer, or else be captured by the $HaveRead$ predicate. The total data is not the exact concatenation of these sequences, since there is potentially an overlap between the retransmission queue and the read buffer, containing data that has been transmitted and acknowledged, but where the acknowledgement has not yet been received by the sender.

  The invariant itself is expressed in classical logic, operating here as a meta-logic for the embedded linear deduction. Specifically, when all the data has been sent, and the read buffer, the retransmission queue and the write buffer are empty, this invariant collapses into a simple statement that the data received is exactly the same as the data that was sent.

- **Mechanically Checking the Specification and Proof**

  To date we have verified manually that this invariant holds for each of our proof rules, and we are currently working on mechanising these theorems using our own implementation of MILL in Isabelle [5]. The verification has been useful in helping to correct some of the axioms (particularly in relation to arithmetic involving the sequence numbers), as well as highlighting various omissions from the presentation in the previous section.

References

[1] V.M. Abrusci. Phase semantics and sequent calculus for pure non-commutative classical linear propositional logic. *Journal of Symbolic Logic*, 56(4):1403–1451, December 1991.

[2] A. Demaille Yet Another Mixed Intuitionistic Linear Logic *Technical Report, École Nationale Supérieure des Télécommunications*, 1998.

[3] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[4] J-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[5] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[6] J. Postel, editor. *RFC 791, Internet Protocol*. Defense Advanced Research Projects Agency, September 1981.

[7] J. Postel, editor. *RFC 793, Transmission Control Protocol*. Defense Advanced Research Projects Agency, September 1981.

[8] D. Sinclair, J. Power, P. Gibson, D. Gray and G. Hamilton. Specifying and Verifying IP with Linear Logic. *International Workshop on Distributed Systems Validation and Verification*, Taipei, Taiwan, April 2000.

Accepted for the 2nd Joint Workshop on Formal Specification of Computer-Based Systems
Washington DC, USA, 20 April, 2001